

# Line Tracking Robot

ECE 3411

Final Project

Brian Marquis  
David Paquette

April 27, 2016

## Contents

<b>1</b>	<b>Objective</b>	<b>2</b>
<b>2</b>	<b>Control Design</b>	<b>2</b>
2.1	Control Overview . . . . .	2
2.2	PID Controller Implementation . . . . .	2
2.3	Path Correction . . . . .	3
2.4	Straight Line Detection and Acceleration . . . . .	4
2.5	Controller Tuning . . . . .	4
<b>3</b>	<b>Sensor Overview</b>	<b>5</b>
3.1	ADC Initialization . . . . .	5
3.2	ADC Channel Switching . . . . .	5
3.3	Reading from the ADC . . . . .	5
<b>4</b>	<b>PWM and Motor Control</b>	<b>6</b>
4.1	PWM Initialization . . . . .	6
4.2	PWM Modulation . . . . .	6
4.3	Motor Control . . . . .	7
<b>5</b>	<b>Conclusions</b>	<b>8</b>
<b>6</b>	<b>Complete Source Code</b>	<b>8</b>
6.1	main.c . . . . .	8
6.2	PIDController.c . . . . .	14
6.3	PIDController.h . . . . .	15

# 1 Objective

The goal of this project is to implement a line follower using the Redbot hardware and the AVR ISA.

## 2 Control Design

The block diagram below shows an overview of our control strategy.

### 2.1 Control Overview

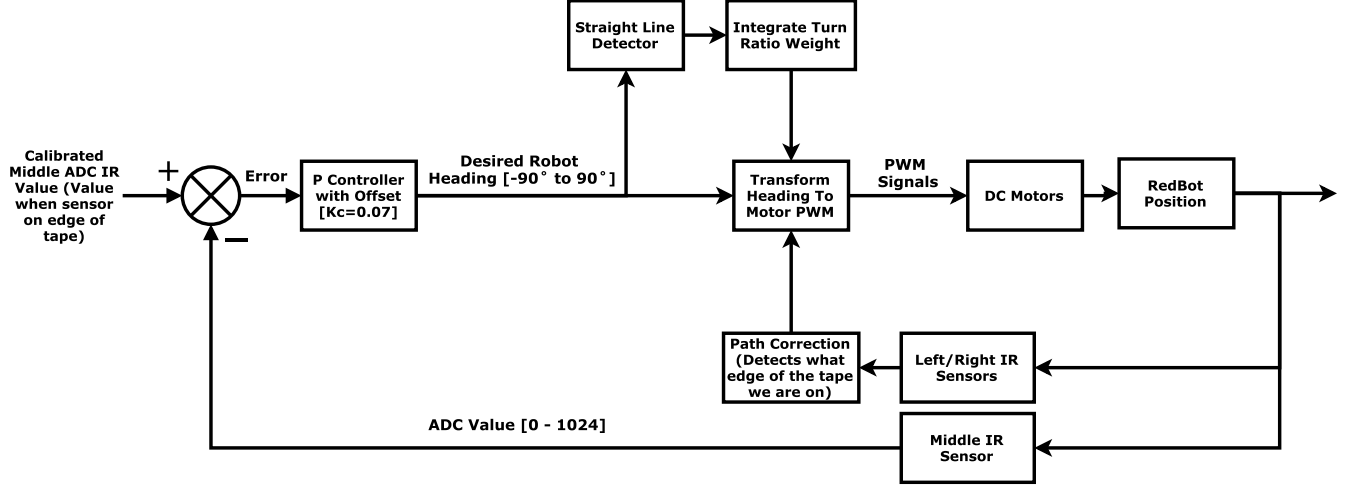


FIG1. Block diagram of controller overview.

### 2.2 PID Controller Implementation

Instead of using a threshold based on/off controller for maintaining our position on the line we used a PID controller. We are using the discrete non-interactive PID control algorithm described by

$$y[n+1] = K_C \left( e[n] + \frac{T_s}{T_I} \sum_n e[n] + T_D \frac{e[n] - e[n-1]}{T_s} \right)$$

Where

$$e[n] = r[n] - m[n]$$

and where  $K_c$  is the controller gain,  $T_I$  is the integral time,  $T_D$  is the derivative time,  $T_s$  is the sampling rate,  $y[n]$  is the controller output,  $r[n]$  is our set point or reference signal and  $m[n]$  is our middle sensor measurement. The implementation can be seen below in code sample 1. To prevent the integral term from continually summing to positive or negative infinity (which in practice would cause the summing variable to overflow back to zero) we included an anti-windup feature in the controller implementation. To prevent unrealistic controller values we saturate the controller output at plus/minus  $90^\circ$ . The  $-90^\circ$  corresponds to a "complete left turn", meaning the left wheel turns off and the right wheel is set to full speed. Plus  $90^\circ$  causes a complete right turn.  $0^\circ$  represents both wheels turning at the same speed, causing the robot to drive in a straight line at the controller's set offset value.

```

1 void PIDControllerComputeOutput(PIDController *controller, float
  processVariable) {
2   //compute error

```

```

3      controller->error = controller->processVariableSetPoint -
        processVariable;
4      //compute integral component
5      if(controller->integralTime) { //prevent divide by zero error
6          controller->errorIntegral = (1 / controller->integralTime) *
            controller->samplingPeriod * controller->error +
            controller->errorIntegral;
7          //wind up protection
8          controller->errorIntegral = controller->errorIntegral <
            controller->minControllerOutput*100 ?
            controller->minControllerOutput*100 :
            controller->errorIntegral;
9          controller->errorIntegral = controller->errorIntegral >
            controller->maxControllerOutput*100 ?
            controller->maxControllerOutput*100 :
            controller->errorIntegral;
10     }
11     //compute derivative term
12     float derivativeComponent =
        (controller->derivativeTime*(controller->previousError -
            controller->error))/controller->samplingPeriod;
13     //compute controller output  $K_c*(e+T_s/T_i*int(e)+T_d*d(e)/T_s)$ 
14     controller->controllerOutput = controller->controllerGain *
        (controller->error + controller->errorIntegral +
            derivativeComponent);
15     controller->previousError = controller->error;
16     controller->controllerOutput = controller->controllerOutput <
        controller->minControllerOutput ? controller->minControllerOutput
        : controller->controllerOutput;
17     controller->controllerOutput = controller->controllerOutput >
        controller->maxControllerOutput ? controller->maxControllerOutput
        : controller->controllerOutput;
18 }

```

**Code sample 1.** (PIDController.c) PID controller compute output implementation with anti-windup and controller output saturation.

## 2.3 Path Correction

The Redbot is designed to keep its middle IR sensor halfway on the tape. If the left or right sensors detect black, indicating that the center has veered off of its mark, the Redbot will align its center sensor to the closest side of the tape. This helps the Redbot to find the path again if it veers off of the path.

```

1      if(direction > 0 && LeftSensor > 700) direction *= -1;
2      if(direction < 0 && RightSensor > 700) direction *= -1;

```

**Code sample 2.** (main.c) Path correction implementation.

If the left or right IR sensor detects black, the direction parameter (1 or -1) will multiply itself by negative one to invert its current direction. When the direction is inverted, the Redbot will now seek to align its middle sensor with the other side of the tape.

## 2.4 Straight Line Detection and Acceleration

The Redbot is designed to keep its middle IR sensor halfway on the tape. If the left or right sensors detect black, indicating that the center has veered off of its mark, the Redbot will align its center sensor to the closest side of the tape. This helps the Redbot to find the path again if it veers off of the path.

```
1 void setLeftMotorDutyCycle(float dutyCycle){
2     Left_duty_cycle = (int)((float)(Left_time_period)*(dutyCycle/100.0));
3 }
4
5 void setRightMotorDutyCycle(float dutyCycle){
6     Right_duty_cycle = (int)((304.0)*(dutyCycle/100.0));
7 }
8
9 void setSpeeds(float error)
10 {
11     if(abs(error) < 11.2){
12         if(turnRatio>0){
13             turnRatio-=0.05;
14         }
15     } else if(abs(error) < 20){
16         if(turnRatio<TURN_POWER) {
17             turnRatio+=0.10;
18         }
19     } else {
20         turnRatio = TURN_POWER;
21     }
22     setLeftMotorDutyCycle(((direction)*-1*error/90.0)*turnRatio+(100.0-turnRatio));
23     setRightMotorDutyCycle(direction*(error/90)*turnRatio+(100.0-turnRatio));
24 }
```

**Code sample 3.** (main.c) Controller heading output to PWM and straight line acceleration implementation. (The parameter *error* is not the controller error, but the controller output, it was poorly named)

The `setLeftMotorDutyCycle` and `setRightMotorDutyCycle` set the duty cycle for each motor. The `setSpeeds` method first checks the previous error value and if it is less than the lower threshold, 11.2, the turn ratio will decrease linearly by 0.05. Decreasing the turn ratio will increase the linear speed and decrease the turn speed. This is used to speed up the Redbot when the path appears to be a straightaway. Otherwise, if the error is greater than 11.2 but less than 20, the turn ratio will increase linearly to provide more power to turn the Redbot. This function is used to gradually slow down the Redbot when it begins to detect a curve. Finally, if the error is greater than 20, the redbot will reset its turn ratio to an optimal value for curved paths. Next, `setLeftMotorDutyCycle` and `setRightMotorDutyCycle` are called using the error and the calculated turn ratio.

## 2.5 Controller Tuning

We tuned the controller using trial and error. The Redbot performed adequately with a  $K_c$  of 0.07. Because the integral and derivative terms introduce more complexity into the trial and error tuning process, we decided to only use a proportional controller. We also tuned the straight line acceleration and turn ratio weights by trial and error.

## 3 Sensor Overview

### 3.1 ADC Initialization

The ADC is initially set to read from channel 6 with a prescaler of 2 so that the conversion is as fast as possible. In addition, the ADC interrupt is enabled so that the active channel can be changed to read other sensors.

```
1 void initADC(){
2     //init the A to D converter
3     ADMUX |= (1<<MUX1)|(1<<MUX2)|(1<< REFS0);
4     ADCSRA = (1<<ADEN) | (1<<ADPS1)|(1<<ADIE);
5 }
```

**Code sample 4.** (main.c) ADC inits.

### 3.2 ADC Channel Switching

A simple state machine is used inside the ADC ISR to change the current sensor to be read. The middle sensor is read first. Next, the left sensor then the right sensor are read.

```
1 ISR( ADC_vect ) {
2     switch(adcChannel){
3         case MIDDLE_SENSOR:
4             ADMUX = 0;
5             ADMUX |= IR_l;
6             adcChannel = LEFT_SENSOR;
7             break;
8         case LEFT_SENSOR:
9             ADMUX = 0;
10            ADMUX |= IR_r;
11            adcChannel = RIGHT_SENSOR;
12            break;
13         case RIGHT_SENSOR:
14             ADMUX = 0;
15             ADMUX |= IR_m;
16             adcChannel = MIDDLE_SENSOR;
17             break;
18     }
19 }
```

**Code sample 5.** (main.c) ADC channel switching.

### 3.3 Reading from the ADC

To read each sensor, the ADC high and low registers are read when the conversion completes. Then the ISR is invoked which changes the current ADC channel and then next read occurs. This process takes place three times in this method to read the middle, left, and right IR sensors with one call.

```
1 float readAnalogVoltage(){
2
3     ADCSRA |= (1 << ADSC);
4     while((ADCSRA & (1<<ADSC)));
5     int adcIn = (ADCL);
6     adcIn |= ( ADCH << 8 );
7 }
```

```

8      ADCSRA |= (1 << ADSC);
9      while((ADCSRA & (1<<ADSC)));
10     RightSensor = (ADCL);
11     RightSensor |= ( ADCH << 8 );
12
13
14     ADCSRA |= (1 << ADSC);
15     while((ADCSRA & (1<<ADSC)));
16     LeftSensor = (ADCL);
17     LeftSensor |= ( ADCH << 8 );
18
19     return adcIn;
20 }

```

**Code sample 6.** (main.c) Reading the ADC.

## 4 PWM and Motor Control

### 4.1 PWM Initialization

Timer 0 is configured to produce a 50kHz PWM signal to OC0B. The duty ratio can be controlled by adjusting the OCR0B register. In addition, Timer 1 is also configured to produce a 50kHz PWM signal. The frequency of timer 1 is controlled by the OCR0A and prescaler values. Finally, the PWM output is handled in the output compare interrupt service routines by toggling the desired pin to drive the motors.

```

1  //Set up Timer 0 for PWM at about 50kHz
2      TCCR0A |= (1<<WGM01)|(1<<WGM00)|(1<<COM0B1); //Fast PWM Mode
3      OCR0B = Left_duty_cycle; //Duty ratio currently at max value 0-255
4      TIMSK0 |= (1<<OCIE0B);
5      TCCR0B |= (1<<CS00); //prescaler of 1
6
7      //Set up Timer 1 for Right Motor PWM
8      TCCR1A |= (1<<WGM10)|(1<<WGM11); //Fast PWM
9      TCCR1B |= ((1<<WGM12)|(1<<WGM13)|(1<<CS10)); // Prescaler = 1
10     TIMSK1 |= ((1<<OCIE1B)|(1<<OCIE1A));
11     OCR1A = Right_time_period;
12     OCR1B = Right_duty_cycle;

```

**Code sample 7.** (main.c) PWM inits.

### 4.2 PWM Modulation

```

1  ISR(TIMER0_COMPB_vect)
2  {
3      OCR0B = Left_duty_cycle;
4  }
5  ISR (TIMER1_COMPA_vect)
6  {
7      OCR1B = Right_duty_cycle;
8      Motor_Bank |= (1<<Right_PWM);
9  }
10
11  ISR (TIMER1_COMPB_vect)

```

```

12 {
13     Motor_Bank &= ~(1<<Right_PWM);
14 }

```

**Code sample 8.** (main.c) PWM modulation.

These ISRs are used to update the duty ratio for each PWM signal and toggle the desired pin to drive the motors.

### 4.3 Motor Control

```

1 void stopMotors()
2 {
3     Motor_Bank &= ~((1<<Left_Mode_1)|(1<<Left_Mode_2)|(1<<Right_Mode_1));
4     //put both motors in stop mode
5     Motor_Bank2 &= ~(1<<Right_Mode_2);
6     Left_duty_cycle = 0;
7     Right_duty_cycle = 0;
8 }
9 void motorForward()
10 {
11     Motor_Bank |= (1<<Left_Mode_2)|(1<<Right_Mode_1);    //put both
12     //motors in forward mode
13     Motor_Bank &= ~(1<<Left_Mode_1);    //put both motors in forward mode
14     Motor_Bank2 &= ~(1<<Right_Mode_2);
15 }
16 void rightForward()
17 {
18     Motor_Bank |= (1<<Right_Mode_1);
19     Motor_Bank2 &= ~(1<<Right_Mode_2);
20 }
21
22 void leftForward()
23 {
24     Motor_Bank |= (1<<Left_Mode_1);
25     Motor_Bank &= ~(1<<Left_Mode_2);
26 }
27
28 void initMotors()
29 {
30     motorForward();
31     Left_duty_cycle = 0; //Left_time_period/5;
32     Right_duty_cycle = 0; //Right_time_period/5;    //MAX SPEED
33 }
34
35 void leftBrake()
36 {
37     Motor_Bank |= (1<<Left_Mode_1)|(1<<Left_Mode_2);
38     Left_duty_cycle = 0;
39 }

```

```

40
41 void rightBrake()
42 {
43     Motor_Bank|=(1<<Right_Mode_1)|(1<<Right_Mode_2);
44     Right_duty_cycle = 0;
45 }
46
47 void leftReverse()
48 {
49     Motor_Bank &= ~((1<<Left_Mode_1));
50     Motor_Bank |= (1<<Left_Mode_2);           //put left motor in reverse mode
51     //keep left PWM the same
52 }
53
54 void rightReverse()
55 {
56     Motor_Bank &= ~((1<<Right_Mode_1));
57     Motor_Bank2 |= (1<<Right_Mode_2);         //put right motor in reverse
58     mode
59     //keep right PWM the same
60 }
61
62 void Reverse()
63 {
64     //put both motors in reverse mode but keep speed the same for now
65     Motor_Bank &= ~((1<<Left_Mode_1)|(1<<Right_Mode_1));
66     Motor_Bank |= (1<<Left_Mode_2)|(1<<Right_Mode_2);

```

**Code sample 9.** (main.c) Motor direction control.

While most of these methods are not used in this source code, they simplify the process of changing the motor modes. These methods simply set the control pins for each motor properly to put each motor in the desired state.

## 5 Conclusions

While our design did follow the lines as intended, it could have handled some of the sharper turns better and the straightaway algorithm could have been optimized. If we had more time, we would have tried to perhaps reverse one wheel when turning to allow for better handling. In addition, we would have tried to use the encoders for the straightaway with a simple controller to keep the angular velocities equal. This would allow the Redbot to travel straight as fast as possible in a straight line.

## 6 Complete Source Code

### 6.1 main.c

```

1  #include <avr/interrupt.h>
2  #define F_CPU 16000000UL /* Tells the Clock Freq to the Compiler. */

```



```

3  #include <avr/io.h> /* Defines pins, ports etc. */
4  #include <stdio.h>
5  #include "PIDController.h"
6  #include <stdlib.h>
7  PIDController motorRatioController;
8
9  volatile int controllerTimer = 0.0;
10 volatile float motorControllerSetpoint = 450.0;
11 volatile float CONTROLLER_GAIN = 0.07; // 0.07 for good line tracking at
    0.55 turn ratio power
12 volatile float CONTROLLER_INTEGRAL_TIME = 0; // 0.15; //seconds
13 volatile float CONTROLLER_DERIVATIVE_TIME = 0; //seconds
14 volatile float CONTROLLER_MIN_OUTPUT = -90.0;
15 volatile float CONTROLLER_MAX_OUTPUT = 90.0;
16 volatile float CONTROLLER_SAMPLING_PERIOD = 0.001;
17 volatile float INITIAL_CONTROLLER_OFFSET = 0.0;
18 volatile float direction = -1;
19 #define Left_PWM 5
20 #define Right_PWM 6
21
22 #define MIDDLE_SENSOR 0
23 #define LEFT_SENSOR 1
24 #define RIGHT_SENSOR 2
25
26
27 #define Left_Mode_1 2
28 #define Right_Mode_1 7
29
30 #define Left_Mode_2 4
31 #define Right_Mode_2 0
32
33 #define Motor_DDR DDRD
34 #define Motor_Bank PORTD
35 #define Motor_DDR2 DDRB
36 #define Motor_Bank2 PORTB
37 // straight values : L = 150 R = 159
38 volatile uint16_t Left_time_period = 255;
39 volatile uint16_t Left_duty_cycle = 80; // 255 max
40
41 volatile uint16_t Right_time_period = 319;
42 volatile uint16_t Right_duty_cycle = 120; // 0-317 (Highest Duty Ratio)
43
44 #define TURN_POWER 55.0 // 40 for line, 55 for arbitrary path
45
46 volatile float turnRatio = TURN_POWER;
47
48 void InitTimer1();
49 void initADC();
50 void setSpeeds(float error);
51
52 #define IR_m (1<<MUX1)|(1<<MUX2)|(1<<REFS0)
53 #define IR_l (1<<MUX1)|(1<<REFS0)
54 #define IR_r (1<<MUX0)|(1<<MUX1)|(1<<REFS0)
55

```

```

56 volatile uint16_t LeftSensor;
57 volatile uint16_t RightSensor;
58
59 volatile int adcChannel = 0;
60
61 void inits(void)
62 {
63     //Set up Data Direction Registers
64     Motor_DDR |=
        (1<<Left_PWM)|(1<<Right_PWM)|(1<<Left_Mode_1)|(1<<Right_Mode_1)|(1<<Left_Mode_
65     Motor_DDR2 |= (1<<Right_Mode_2);
66
67     //Set up Timer 0 for PWM at about 50kHz
68     TCCR0A |= (1<<WGM01)|(1<<WGM00)|(1<<COM0B1); //Fast PWM Mode
69     OCR0B = Left_duty_cycle; //Duty ratio currently at max value 0-255
70     TIMSK0 |= (1<<OCIE0B);
71     TCCR0B |= (1<<CS00); //prescaler of 1
72
73     //Set up Timer 1 for Right Motor PWM
74     TCCR1A |= (1<<WGM10)|(1<<WGM11); //Fast PWM
75     TCCR1B |= ((1<<WGM12)|(1<<WGM13)|(1<<CS10)); // Prescaler = 1
76     TIMSK1 |= ((1<<OCIE1B)|(1<<OCIE1A));
77     OCR1A = Right_time_period;
78     OCR1B = Right_duty_cycle;
79
80
81     //Set up Timer 2 as a 1ms clock
82     TCCR2A |= (1<<WGM21); //CTC Mode
83     OCR2A = 249;
84     TIMSK2 |= (1<<OCIE2A);
85     TCCR2B |= (1<<CS22); //64 prescaler */
86     initADC();
87     motorRatioController = PIDControllerCreate(motorControllerSetpoint,
88     CONTROLLER_GAIN, CONTROLLER_INTEGRAL_TIME,
89     CONTROLLER_DERIVATIVE_TIME,
90     CONTROLLER_MIN_OUTPUT, CONTROLLER_MAX_OUTPUT,
91     CONTROLLER_SAMPLING_PERIOD,
92     INITIAL_CONTROLLER_OFFSET);
93     sei();
94 }
95
96 ISR( ADC_vect ) {
97     switch(adcChannel){
98     case MIDDLE_SENSOR:
99         ADMUX = 0;
100         ADMUX |= IR_l;
101         adcChannel = LEFT_SENSOR;
102         break;
103     case LEFT_SENSOR:
104         ADMUX = 0;
105         ADMUX |= IR_r;
106         adcChannel = RIGHT_SENSOR;
107         break;
108     case RIGHT_SENSOR:

```

```

107         ADMUX = 0;
108         ADMUX |= IR_m;
109         adcChannel = MIDDLE_SENSOR;
110         break;
111     }
112 }
113
114 float readAnalogVoltage(){
115     ADCSRA |= (1 << ADSC);
116     while((ADCSRA & (1<<ADSC)));
117     int adcIn = (ADCL);
118     adcIn |= ( ADCH << 8 );
119
120     ADCSRA |= (1 << ADSC);
121     while((ADCSRA & (1<<ADSC)));
122     RightSensor = (ADCL);
123     RightSensor |= ( ADCH << 8 );
124
125     ADCSRA |= (1 << ADSC);
126     while((ADCSRA & (1<<ADSC)));
127     LeftSensor = (ADCL);
128     LeftSensor |= ( ADCH << 8 );
129
130     return adcIn;
131 }
132
133 ISR(TIMER2_COMPA_vect) {
134     controllerTimer++;
135 }
136
137 void initADC(){
138     //init the A to D converter
139     ADMUX |= (1<<MUX1)|(1<<MUX2) |(1<< REFS0);
140     ADCSRA = (1<<ADEN) | (1<<ADPS1)|(1<<ADIE);
141 }
142
143 ISR(TIMER0_COMPB_vect)
144 {
145     OCR0B = Left_duty_cycle;
146 }
147
148 ISR (TIMER1_COMPA_vect)
149 {
150     OCR1B = Right_duty_cycle;
151     Motor_Bank |= (1<<Right_PWM);
152 }
153
154 ISR (TIMER1_COMPB_vect)
155 {
156     Motor_Bank &= ~(1<<Right_PWM);
157 }
158
159 void setLeftMotorDutyCycle(float dutyCycle){

```

```

161     Left_duty_cycle = (int)((float)(Left_time_period)*(dutyCycle/100.0));
162 }
163
164 void setRightMotorDutyCycle(float dutyCycle){
165     Right_duty_cycle = (int)((304.0)*(dutyCycle/100.0));
166 }
167
168 void setSpeeds(float error)
169 {
170     if(abs(error) < 11.2){
171         if(turnRatio>0){
172             turnRatio-=0.05;
173         }
174     } else if(abs(error) < 20){
175         if(turnRatio<TURN_POWER) {
176             turnRatio+=0.10;
177         }
178     } else {
179         turnRatio = TURN_POWER;
180     }
181     setLeftMotorDutyCycle(((direction)*-1*error/90.0)*turnRatio+(100.0-turnRatio));
182     setRightMotorDutyCycle(direction*(error/90)*turnRatio+(100.0-turnRatio));
183 }
184
185 void stopMotors()
186 {
187     Motor_Bank &= ~(1<<Left_Mode_1)|(1<<Left_Mode_2)|(1<<Right_Mode_1));
188     //put both motors in stop mode
189     Motor_Bank2 &= ~(1<<Right_Mode_2);
190     Left_duty_cycle = 0;
191     Right_duty_cycle = 0;
192 }
193
194 void motorForward()
195 {
196     Motor_Bank |= (1<<Left_Mode_2)|(1<<Right_Mode_1);    //put both
197     //motors in forward mode
198     Motor_Bank &= ~(1<<Left_Mode_1);    //put both motors in forward mode
199     Motor_Bank2 &= ~(1<<Right_Mode_2);
200 }
201
202 void rightForward()
203 {
204     Motor_Bank |= (1<<Right_Mode_1);
205     Motor_Bank2 &= ~(1<<Right_Mode_2);
206 }
207
208 void leftForward()
209 {
210     Motor_Bank |= (1<<Left_Mode_1);
211     Motor_Bank &= ~(1<<Left_Mode_2);
212 }
213
214 void initMotors()

```

```

213 {
214     motorForward();
215     Left_duty_cycle = 0; //Left_time_period/5;
216     Right_duty_cycle = 0; //Right_time_period/5;           //MAX SPEED
217 }
218
219 void leftBrake()
220 {
221     Motor_Bank|=(1<<Left_Mode_1)|(1<<Left_Mode_2);
222     Left_duty_cycle = 0;
223 }
224
225 void rightBrake()
226 {
227     Motor_Bank|=(1<<Right_Mode_1)|(1<<Right_Mode_2);
228     Right_duty_cycle = 0;
229 }
230
231 void leftReverse()
232 {
233     Motor_Bank &= ~((1<<Left_Mode_1));
234     Motor_Bank |= (1<<Left_Mode_2);           //put left motor in reverse mode
235     //keep left PWM the same
236 }
237
238 void rightReverse()
239 {
240     Motor_Bank &= ~((1<<Right_Mode_1));
241     Motor_Bank2 |= (1<<Right_Mode_2);         //put right motor in reverse
242     mode
243     //keep right PWM the same
244 }
245
246 void Reverse()
247 {
248     //put both motors in reverse mode but keep speed the same for now
249     Motor_Bank &= ~((1<<Left_Mode_1)|(1<<Right_Mode_1));
250     Motor_Bank |= (1<<Left_Mode_2)|(1<<Right_Mode_2);
251 }
252
253 int main(void)
254 {
255     inits();
256     leftForward();
257     rightReverse();
258     while(1){
259         if(controllerTimer > motorRatioController.samplingPeriod * 1000){
260             float middleSensorValue = readAnalogVoltage();
261             if(direction > 0 && LeftSensor > 700) direction *= -1;
262             if(direction < 0 && RightSensor > 700) direction *= -1;
263             PIDControllerComputeOutput(&motorRatioController,
264                 middleSensorValue);
265             setSpeeds(motorRatioController.controllerOutput);
266             controllerTimer = 0;

```

```

265     }
266 }
267 return 0;
268 }

```

**Code sample 10.** (main.c) Main source file

## 6.2 PIDController.c

```

1  #include "PIDController.h"
2
3
4  PIDController PIDControllerCreate(float processVariableSetPoint, float
    controllerGain,
5      float integralTime, float derivativeTime, float
        minControllerOutput, float maxControllerOutput, float
        samplingPeriod, float initialControllerOffset) {
6      PIDController newController;
7      newController.processVariableSetPoint = processVariableSetPoint;
8      newController.controllerGain = controllerGain;
9      newController.integralTime = integralTime;
10     newController.derivativeTime = derivativeTime;
11     newController.error = 0;
12     newController.previousError = 0;
13     newController.errorIntegral = initialControllerOffset;
14     newController.controllerOutput = 0;
15     newController.minControllerOutput = minControllerOutput;
16     newController.maxControllerOutput = maxControllerOutput;
17     newController.samplingPeriod = samplingPeriod;
18     return newController;
19 }
20
21 void PIDControllerComputeOutput(PIDController *controller, float
    processVariable) {
22     //compute error
23     controller->error = controller->processVariableSetPoint -
        processVariable;
24     //compute integral component
25     if(controller->integralTime) { //prevent divide by zero error
26         controller->errorIntegral = (1 / controller->integralTime) *
            controller->samplingPeriod * controller->error +
            controller->errorIntegral;
27     }
28     //wind up protection
29     controller->errorIntegral = controller->errorIntegral <
        controller->minControllerOutput*100 ?
        controller->minControllerOutput*100 :
        controller->errorIntegral;
30     controller->errorIntegral = controller->errorIntegral >
        controller->maxControllerOutput*100 ?
        controller->maxControllerOutput*100 :
        controller->errorIntegral;
31 }
32 //compute derivative term
    float derivativeComponent =

```

```

        (controller->derivativeTime*(controller->previousError -
        controller->error))/controller->samplingPeriod;
33 //compute controller output  $K_c*(e+Ts/T_i*int(e)+T_d*d(e)/Ts)$ 
34 controller->controllerOutput = controller->controllerGain *
        (controller->error + controller->errorIntegral +
        derivativeComponent);
35 controller->previousError = controller->error;
36 controller->controllerOutput = controller->controllerOutput <
        controller->minControllerOutput ? controller->minControllerOutput
        : controller->controllerOutput;
37 controller->controllerOutput = controller->controllerOutput >
        controller->maxControllerOutput ? controller->maxControllerOutput
        : controller->controllerOutput;
38 }

```

**Code sample 11.** (PIDController.c) PID controller source file.

### 6.3 PIDController.h

```

1  #ifndef PIDCONTROLLER
2  #define PIDCONTROLLER
3
4  typedef struct PIDController {
5      float processVariableSetPoint;
6      float controllerGain;
7      float integralTime;
8      float derivativeTime;
9      float error;
10     float previousError;
11     float errorIntegral;
12     float minControllerOutput;
13     float maxControllerOutput;
14     float samplingPeriod;
15     float controllerOutput;
16 } PIDController;
17
18 PIDController PIDControllerCreate(float processVariableSetPoint, float
    controllerGain,
19     float integralTime, float derivativeTime, float
        minControllerOutput, float maxControllerOutput, float
        samplingPeriod, float initialControllerOffset);
20
21 void PIDControllerComputeOutput(PIDController *controller, float
    processVariable);
22
23 #endif

```

**Code sample 12.** (PIDController.h) Header file for PIDController.c